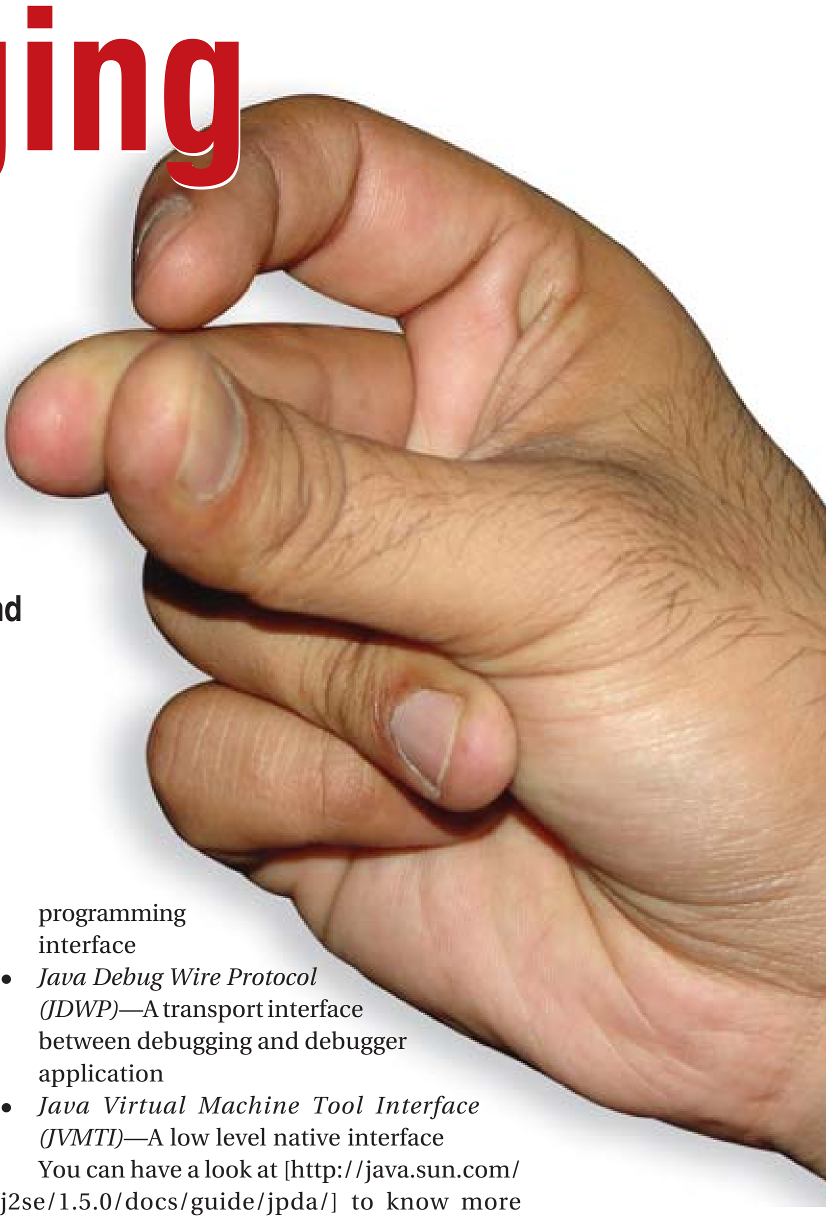
JCodeFlow Debugging Made Simple



Trace Java programs with the simple and lightweight Java Platform Debugger Architecture-based JCodeFlow tool!

In the December 2004 LFY issue, we introduced you to a CodeFlow tool, which can be used to trace programs written in C/C++. In this article, we will explore a similar tool called JCodeFlow, which can be used to trace Java programs.

JCodeFlow is a simple, lightweight Java Platform Debugger Architecture (JPDA)-based tool that can be used to trace any program written in Java. It uses the JPDA *MethodEntryEvent* and *MethodExitEvent* to get the code flow (trace) data. By using this tool, you can easily understand the flow of a program in no time.

JPDA provides the necessary infrastructure needed to build end-user debugger applications like JCodeFlow. The architecture of JPDA is shown in Figure 1.

JPDA provides three interfaces to the user:

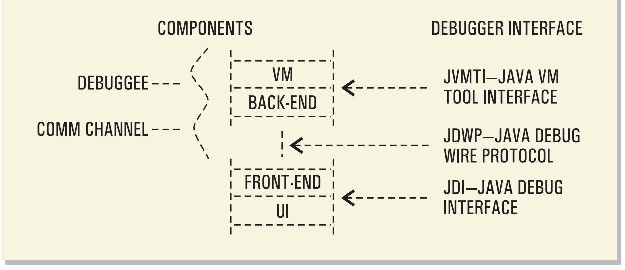
- *Java Debug Interface (JDI)*—A high level Java programming interface
- *Java Debug Wire Protocol (JDWP)*—A transport interface between debugging and debugger application
- *Java Virtual Machine Tool Interface (JVMTI)*—A low level native interface

You can have a look at [http://java.sun.com/j2se/1.5.0/docs/guide/jpda/] to know more about JPDA architecture. The JCodeFlow tool is based on the JDI interface of JPDA architecture. It is completely written in Java and uses the default implementation of JPDA interfaces to get the trace data.

This tool has been designed in such a way that it can connect to the remote Java Virtual Machine (VM) running a Java application.

Listing 1 shows the main entry function to JCodeFlow tool.

Figure 1: JPDA architecture



Listing 1: JCodeFlow entry function

```
public void doCodeFlow() throws Throwable {
    Connector con = null;
    Map conArgs = null;
    VirtualMachine vm = null;
    EventProcessor ep = null;

    try {
        con = getConnector("com.sun.jdi.SocketAttach");
        conArgs = getConnectorArgs(con);

        setConnectorArg(conArgs,"hostname","localhost");
        setConnectorArg(conArgs,"port","4000");

        vm = connectToRemoteVM(con,conArgs);

        ep = new EventProcessor(vm);
        ep.start();

        enableEvents(vm);

        vm.resume();
    } catch(Throwable t) {
        throw t;
    }
}

public static void main(String args[])
{
    JCodeFlow jcodeFlow = null;
    try {
        jcodeFlow = new JCodeFlow();
        jcodeFlow.doCodeFlow();
    } catch(Throwable t) {
        t.printStackTrace();
    }
}
```

The job of the *doCodeFlow* function is to connect to the remote VM using *AttachingConnector*, enable events and start event processing thread. The function *getConnector* is used to get the *AttachingConnector* to connect to the remote running VM. It uses the initial Bootstrap class to access the default JDI implementation. The required arguments (hostname and port) for *AttachingConnector* are set with the *setConnectorArg* function. The value 'localhost' for the hostname parameter indicates that the debugger and debugging application are running on the same machine. The *connectToRemoteVM* connects the debugger to remote debugging application. Listing 2 shows the implementation of these functions.

Listing 2: JCodeFlow connection helper functions

```
private VirtualMachine connectToRemoteVM(Connector con, Map
conArgs) throws Throwable {
    try {
        return( ((AttachingConnector) con).attach(conArgs) );
    } catch(Throwable t) {
        throw t;
    }
}

private void setConnectorArg(Map conArgs, String argName,
String argValue) throws Throwable {
    Iterator conArgsItr = null;
    Connector.Argument arg = null;

    try {
        conArgsItr = conArgs.keySet().iterator();
        while( conArgsItr.hasNext() ) {
            arg = (Connector.Argument) conArgs.get((String)
conArgsItr.next());
            if( arg.name().equals(argName) ) {
                arg.setValue(argValue);
                conArgs.put(argName,arg);
                return;
            }
        }

        throw new Exception("'" + argName + "' is invalid.");
    } catch(Throwable t) {
        throw t;
    }
}

private Map getConnectorArgs(Connector con) throws Throwable {
    try {
        return( (Map) con.defaultArguments() );
    } catch(Throwable t) {
        throw t;
    }
}

private  Connector  getConnector(String  conName)  throws  Throwable  {
    Iterator conItr = null;
    Connector con = null;

    try {
        conItr =
Bootstrap.virtualMachineManager().allConnectors().iterator();
        while( conItr.hasNext() ) {
            con = (Connector) conItr.next();
            if( con.name().equals(conName) ) {
                return con;
            }
        }

        throw new Exception("'" + conName + "' is invalid.");
    } catch(Throwable t) {
        throw t;
    }
}
```

The *MethodEntryEvent* and *MethodExitEvents* are enabled in the *enableEvents* function. Since we are not interested in getting the code flow for Java system classes, we exclude them through the *addClassExclusionFilter* method as shown in Listing 3.

Listing 3: JCodeFlow event enable helper functions

```
private String[] classExcludes = { "java.*", "javax.*",
"sun.*", "com.sun.*" };

private void enableMethodExitEvent(EventRequestManager erMgr)
throws Throwable {
    MethodExitRequest meReq = null;
    int i = 0;

    try {
        meReq = erMgr.createMethodExitRequest();

        for( i=0; i<classExcludes.length; i++ ) {
            meReq.addClassExclusionFilter(classExcludes[i]);
        }

        meReq.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        meReq.enable();
    } catch(Throwable t) {
        throw t;
    }
}

private void enableMethodEntryEvent(EventRequestManager erMgr)
throws Throwable {
    MethodEntryRequest meReq = null;
    int i = 0;

    try {
        meReq = erMgr.createMethodEntryRequest();

        for( i=0; i<classExcludes.length; i++ ) {
            meReq.addClassExclusionFilter(classExcludes[i]);
        }

        meReq.setSuspendPolicy(EventRequest.SUSPEND_NONE);
        meReq.enable();
    } catch(Throwable t) {
        throw t;
    }
}

private void enableEvents(VirtualMachine vm) throws Throwable {
    EventRequestManager erMgr = null;


    try {
        erMgr = vm.eventRequestManager();

        enableMethodEntryEvent(erMgr);
        enableMethodExitEvent(erMgr);
    } catch(Throwable t) {
        throw t;
    }
}
```

JCodeFlow uses a separate thread (*EventProcessor*) to handle the events generated by remote VM. The main job of the *EventProcessor* thread is to process the *MethodEntryEvent* and *MethodExitEvent*, to get the code flow data. Listing 4 shows the implementation of the thread's run method.

Listing 4: JCodeFlow event-process thread function

```
private void handleEvent(Event event) throws Throwable {
    try {
        if( event instanceof MethodEntryEvent ) {
            handleMethodEntryEvent((MethodEntryEvent) event);
            return;
        }

        if( event instanceof MethodExitEvent ) {
            handleMethodExitEvent((MethodExitEvent) event);
            return;
        }

        if( event instanceof VMStartEvent ) {
            System.out.println("Remote VM started.");
            return;
        }

        if( (event instanceof VMDisconnectEvent) || (event
instanceof VMDeathEvent) ) {
            System.out.println("Remote VM disconnected.");
            stop = true;
            return;
        }

        throw new Exception("Unexpected event received.");
    } catch(Throwable t) {
        throw t;
    }
}

public void run() {
    EventQueue eventQ = null;
    EventSet eventSet = null;
    EventIterator eventItr = null;

    try {
        eventQ = vm.eventQueue();

        while(!stop) {
            try {
                eventSet = eventQ.remove();
                eventItr = eventSet.eventIterator();

                while( eventItr.hasNext() ) {
                    handleEvent(eventItr.nextEvent());
                }

                eventSet.resume();
                try {
                    Thread.yield();
                } catch(Throwable ignore) {
                    //Ignore
                }
            } catch(Throwable t) {
                t.printStackTrace();
                break;
            }
        }
    } catch(Throwable t) {
        eventSet.resume();
    }
}
```

EventQueue is the manager of incoming debugger events for a target VM. Several events may be created at a given time by the target VM. All those events are delivered together as an *EventSet*. JCodeFlow processes one event at a time. Listing 5 shows the JCodeFlow event handling functions.

```
Listing 5: JCodeFlow event handling functions

private void handleMethodExitEvent(MethodExitEvent event)
throws Throwable {
    try {
        Method method = event.method();

        if( method.isSynthetic() || method.isAbstract()  ||
method.isNative() ) {
            return;
        }

        indent–;
    } catch(Throwable t) {
        throw t;
    }
}

private void handleMethodEntryEvent(MethodEntryEvent event)
throws Throwable {
    try {
        Method method = event.method();

        if( method.isSynthetic() || method.isAbstract()  ||
method.isNative() ) {
            return;
        }

        for( long i=0; i<indent; i++ ) { System.out.print("\t"); }

        System.out.println(method.name() + "(" +
(method.location().lineNumber()) + ", " +
            method.location().sourceName() + ")");

        indent++;
    } catch(Throwable t) {
        throw t;
    }
}
```

MethodEntry and *MethodExit* events are handled by the *handleMethodEntryEvent* and *handleMethodExitEvent* function respectively. Nothing much is done in these functions. We just increment the indent value and display the method details in MethodEntryEvent and deduct the indent value in the *MethodExitEvent*. The indent value is used to show the details in tree-like format. The complete JCodeFlow source is included in the LFY CD.

To demonstrate the power of JCodeFlow, we will use the JEmailClient program incorporated in the LFY CD. Copy the two Java files *JEmailClientInt.java* and *JEmailClient.java* from the CD-ROM to your system hard disk and compile them as shown...

```
javac JEmailClientInt.java
javac JEmailClient.java
```

Copy the *JCodeFlow.java* from the CD-ROM to your system hard disk and compile the program as indicated...

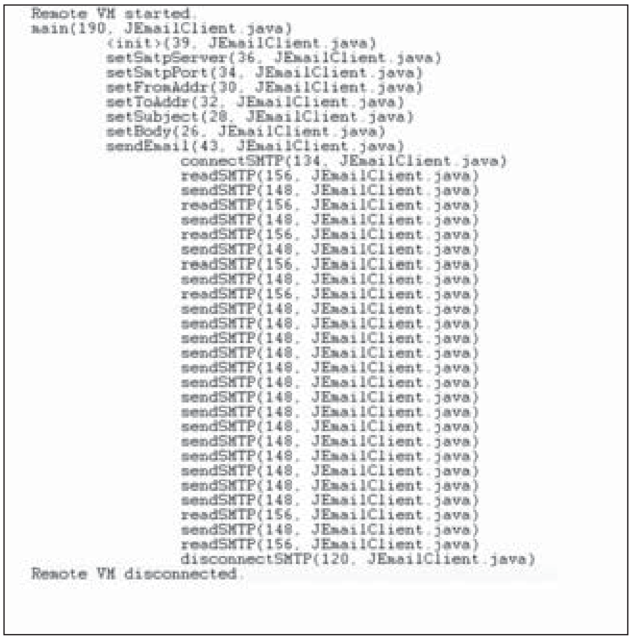
```
javac JCodeFlow.java
```

Start the JEmailClient program

```
java -
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=4000 -
Xdebug JEmailClient
```

Then start the JCodeFlow program.

```
java -cp <path_to_java>/lib/tools.jar;. JCodeFlow
```



```
Remote VM started.
main(190, JEmailClient.java)
        <init>(39, JEmailClient.java)
        setSmtpServer(36, JEmailClient.java)
        setSmtpPort(34, JEmailClient.java)
        setFromAddr(30, JEmailClient.java)
        setToAddr(32, JEmailClient.java)
        setSubject(28, JEmailClient.java)
        setBody(26, JEmailClient.java)
        sendEmail(43, JEmailClient.java)
                connectSMTP(134, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                sendSMTP(148, JEmailClient.java)
                readSMTP(156, JEmailClient.java)
                disconnectSMTP(120, JEmailClient.java)
Remote VM disconnected.
```

Figure 2: JCodeFlow in action

Figure 2 shows the final program flow (call graph) report for a sample e-mail client program. You can now see how easy it is to understand the Java program with our simple and lightweight JCodeFlow tool. Happy debugging!

References

- JDI—http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/index.html?overview-summary.html
- JPDA—http://java.sun.com/products/jpda/

LFY

By: Raja R K. The author is a lead engineer working with HCL Technologies (Cisco Systems Offshore Development Centre) in Chennai. He can be contacted at: rajark_hcl@yahoo.co.in